1 Getting started with ASP.NET Core

This chapter covers

- What is ASP.NET Core?
- Things you can build with ASP.NET Core
- The advantages and limitations of .NET Core and .NET 5.0
- How ASP.NET Core works

Choosing to learn and develop with a new framework is a big investment, so it's important to establish early on whether it's right for you. In this chapter, I provide some background about ASP.NET Core: what it is, how it works, and why you should consider it for building your web applications.

If you're new to .NET development, this chapter will help you to understand the .NET landscape. For existing .NET developers, I provide guidance on whether now is the right time to consider moving your focus to .NET Core and .NET 5.0, and on the advantages ASP.NET Core can offer over previous versions of ASP.NET.

By the end of this chapter, you should have a good overview of the .NET landscape, the role of .NET 5.0, and the basic mechanics of how ASP.NET Core works—so without further ado, let's dive in!

1.1 An introduction to ASP.NET Core

ASP.NET Core is a cross-platform, open source, web application framework that you can use to quickly build dynamic, server-side rendered applications. You can also use ASP.NET Core to create HTTP APIs that can be consumed by mobile applications, by browser-based single-page applications such as Angular and React, or by other back-end applications.

ASP.NET Core provides structure, helper functions, and a framework for building applications, which saves you having to write a lot of this code yourself. The ASP.NET Core framework code then calls into your "handlers" which, in turn, call methods in your application's business logic, as shown in figure 1.1. This business logic is the core of your application. You can interact with other services here, such as databases or remote APIs, but your business logic does not typically depend *directly* on ASP.NET Core.

In this section I cover

- The reasons for using a web framework
- The previous ASP.NET framework's benefits and limitations
- What ASP.NET Core is and its motivations

At the end of this section you should have a good sense of why ASP.NET Core was created, its design goals, and why you might want to use it.

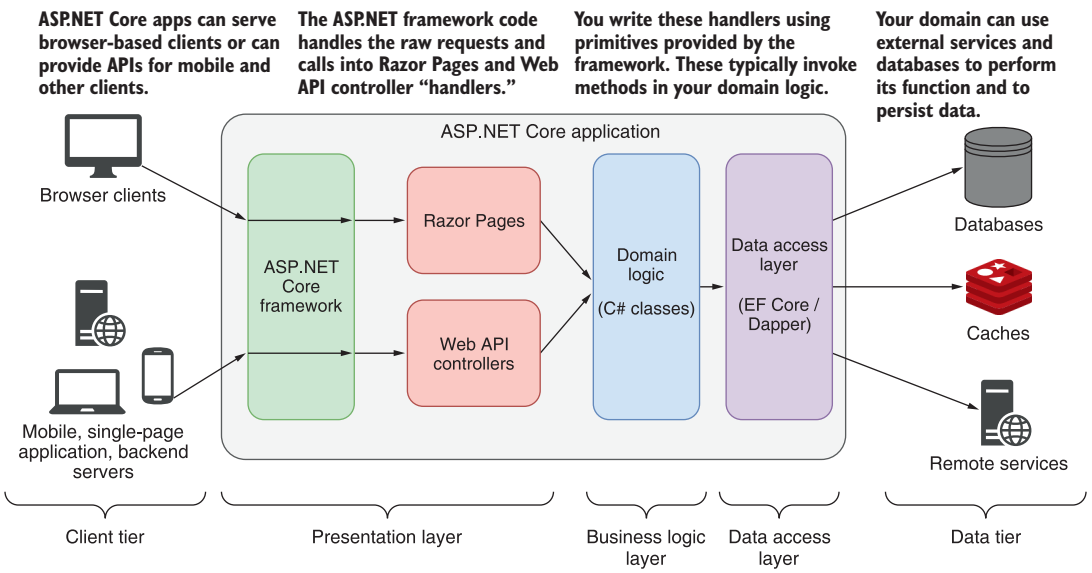

Figure 1.1 A typical ASP.NET Core application consists of several layers. The ASP.NET Core framework code handles requests from a client, dealing with the complex networking code. The framework then calls into handlers (Razor Pages and Web API controllers) that you write using primitives provided by the framework. Finally, these handlers call into your application's domain logic, which are typically C# classes and objects without any ASP.NET Core-specific dependencies.

1.1.1 Using a web framework

If you're new to web development, it can be daunting moving into an area with so many buzzwords and a plethora of ever-changing products. You may be wondering if they're all necessary—how hard can it be to return a file from a server?

Well, it's perfectly possible to build a static web application without the use of a web framework, but its capabilities will be limited. As soon as you want to provide any kind of security or dynamism, you'll likely run into difficulties, and the original simplicity that enticed you will fade before your eyes.

Just as desktop or mobile development frameworks can help you build native applications, ASP.NET Core makes writing web applications faster, easier, and more secure than trying to build everything from scratch. It contains libraries for common things like

- Creating dynamically changing web pages
- Letting users log in to your web app
- Letting users use their Facebook account to log in to your web app using OAuth
- Providing a common structure for building maintainable applications
- Reading configuration files
- Serving image files
- Logging requests made to your web app

The key to any modern web application is the ability to generate dynamic web pages. A *dynamic web page* may display different data depending on the current logged-in user, or it could display content submitted by users. Without a dynamic framework, it wouldn't be possible to log in to websites or to display any sort of personalized data on a page. In short, websites like Amazon, eBay, and Stack Overflow (seen in figure 1.2) wouldn't be possible.

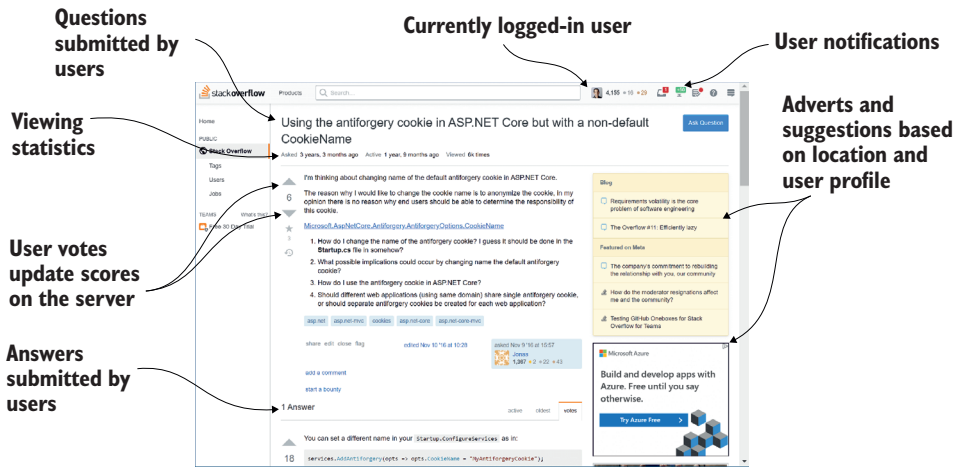

Figure 1.2 The Stack Overflow website (https://stackoverflow.com) is built using ASP.NET and is almost entirely dynamic content.

The benefits and limitations of ASP.NET

ASP.NET Core is the latest evolution of Microsoft's popular ASP.NET web framework, released in June 2016. Previous versions of ASP.NET have seen many incremental updates, focusing on high developer productivity and prioritizing backwards compatibility. ASP.NET Core bucks that trend by making significant architectural changes that rethink the way the web framework is designed and built.

ASP.NET Core owes a lot to its ASP.NET heritage, and many features have been carried forward from before, but ASP.NET Core is a new framework. The whole technology stack has been rewritten, including both the web framework and the underlying platform.

At the heart of the changes is the philosophy that ASP.NET should be able to hold its head high when measured against other modern frameworks, but that existing .NET developers should continue to have a sense of familiarity.

To understand *why* Microsoft decided to build a new framework, it's important to understand the benefits and limitations of the previous ASP.NET web framework.

The first version of ASP.NET was released in 2002 as part of .NET Framework 1.0, in response to the then conventional scripting environments of classic ASP and PHP. ASP.NET Web Forms allowed developers to rapidly create web applications using a graphical designer and a simple event model that mirrored desktop application-building techniques.

The ASP.NET framework allowed developers to quickly create new applications, but over time the web development ecosystem changed. It became apparent that ASP.NET Web Forms suffered from many issues, especially when building larger applications. In particular, a lack of testability, a complex stateful model, and limited influence over the generated HTML (making client-side development difficult) led developers to evaluate other options.

In response, Microsoft released the first version of ASP.NET MVC in 2009, based on the Model-View-Controller pattern, a common web design pattern used in other frameworks such as Ruby on Rails, Django, and Java Spring. This framework allowed you to separate UI elements from application logic, made testing easier, and provided tighter control over the HTML-generation process.

ASP.NET MVC has been through four more iterations since its first release, but they have all been built on the same underlying framework provided by the System.Web.dll file. This library is part of .NET Framework, so it comes pre-installed with all versions of Windows. It contains all the core code that ASP.NET uses when you build a web application.

This dependency brings both advantages and disadvantages. On the one hand, the ASP.NET framework is a reliable, battle-tested platform that's fine for building web applications on Windows. It provides a wide range of features, which have seen many years in production, and it is well known by virtually all Windows web developers.

On the other hand, this reliance is limiting—changes to the underlying System.Web.dll file are far-reaching and, consequently, slow to roll out. This limits the extent to which

ASP.NET is free to evolve and results in release cycles only happening every few years. There's also an explicit coupling with the Windows web host, Internet Information Service (IIS), which precludes its use on non-Windows platforms.

More recently, Microsoft has declared .NET Framework to be "done." It won't be removed or replaced, but it also won't receive any new features. Consequently, ASP.NET based on System.Web.dll will not receive new features or updates either.

In recent years, many web developers have started looking at cross-platform web frameworks that can run on Windows as well as Linux and macOS. Microsoft felt the time had come to create a framework that was no longer tied to its Windows legacy, and thus ASP.NET Core was born.

1.1.2 What is ASP.NET Core?

The development of ASP.NET Core was motivated by the desire to create a web framework with four main goals:

- To be run and developed cross-platform
- To have a modular architecture for easier maintenance
- To be developed completely as open source software
- To be applicable to current trends in web development, such as client-side applications and deployment to cloud environments

To achieve all these goals, Microsoft needed a platform that could provide underlying libraries for creating basic objects such as lists and dictionaries, and for performing, for example, simple file operations. Up to this point, ASP.NET development had always been focused, and dependent, on the Windows-only .NET Framework. For ASP.NET Core, Microsoft created a lightweight platform that runs on Windows, Linux, and macOS called .NET Core (and subsequently .NET 5.0), as shown in figure 1.3.

DEFINITION .NET 5.0 is the next version of .NET Core after 3.1. It represents a unification of .NET Core and other .NET platforms into a single runtime and framework. The terms .NET Core and .NET 5.0 are often used interchangeably, but for consistency with Microsoft's language, I use the term .NET 5.0 to refer to the latest version of .NET Core, and .NET Core when referring to previous versions.

.NET Core (and its successor, .NET 5.0) employs many of the same APIs as .NET Framework, but it's more modular and only implements a subset of the features .NET Framework does, with the goal of providing a simpler implementation and programming model. It's a completely separate platform, rather than a fork of .NET Framework, though it uses similar code for many of its APIs.

With .NET 5.0 alone, it's possible to build console applications that run cross-platform. Microsoft created ASP.NET Core to be an additional layer on top of console applications, such that converting to a web application involves adding and composing libraries, as shown in figure 1.4.

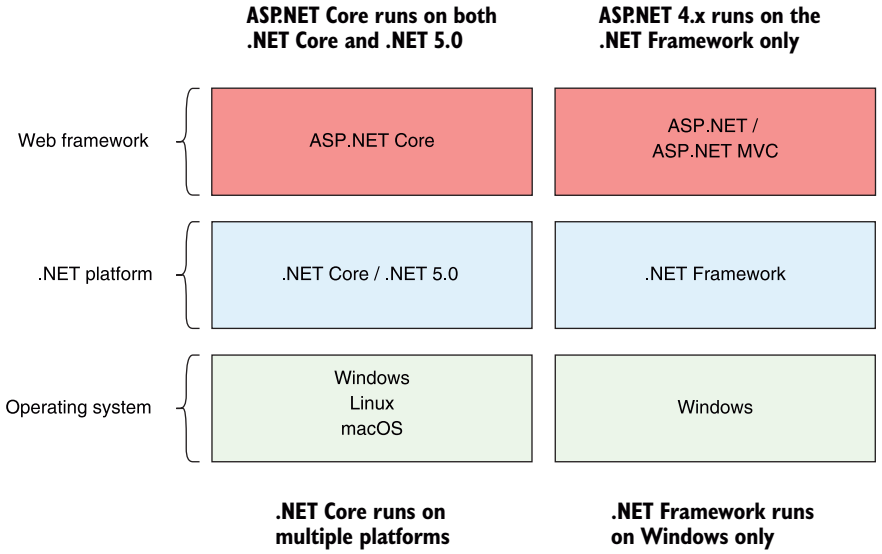

Figure 1.3 The relationship between ASP.NET Core, ASP.NET, .NET Core/.NET 5.0, and .NET Framework. ASP.NET Core runs on .NET Core and .NET 5.0, so it can run cross-platform. Conversely, ASP.NET runs on .NET Framework only, so it is tied to the Windows OS.

You write a .NET 5.0 console app that starts up an instance of an ASP.NET Core web server.

Microsoft provides a cross-platform web server by default, called Kestrel.

Your web application logic is run by Kestrel. You'll use various libraries to enable features such as logging and HTML generation as required.

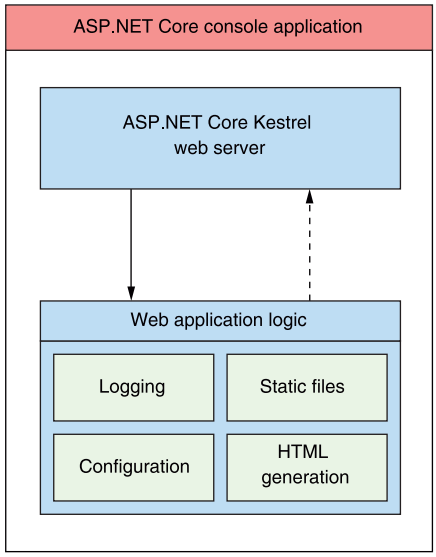

Figure 1.4 The ASP.NET Core application model. The .NET 5.0 platform provides a base console application model for running command-line apps. Adding a web server library converts this into an ASP.NET Core web app. Additional features, such as configuration and logging, are added by way of additional libraries.

By adding an ASP.NET Core web server to your .NET 5.0 app, your application can run as a web application. ASP.NET Core is composed of many small libraries that you can choose from to provide your application with different features. You'll rarely need all the libraries available to you, and you only add what you need. Some of the libraries are common and will appear in virtually every application you create, such as the ones for reading configuration files or performing logging. Other libraries build on top of these base capabilities to provide application-specific functionality, such as third-party logging-in via Facebook or Google.

Most of the libraries you'll use in ASP.NET Core can be found on GitHub, in the Microsoft ASP.NET Core organization repositories at https://github.com/dotnet/aspnetcore. You can find the core libraries there, such as the authentication and logging libraries, as well as many more peripheral libraries, such as the third-party authentication libraries.

All ASP.NET Core applications will follow a similar design for basic configuration, as suggested by the common libraries, but in general the framework is flexible, leaving you free to create your own code conventions. These common libraries, the extension libraries that build on them, and the design conventions they promote are covered by the somewhat nebulous term ASP.NET Core.

1.2 When to choose ASP.NET Core

Hopefully, you now have a general grasp of what ASP.NET Core is and how it was designed. But the question remains: should you use it? Microsoft is recommending that all new .NET web development should use ASP.NET Core, but switching to or learning a new web stack is a big ask for any developer or company. In this section I cover

- What sort of applications you can build with ASP.NET Core
- Some of the highlights of ASP.NET Core
- Why you should consider using ASP.NET Core for new applications
- Things to consider before converting existing ASP.NET applications to ASP.NET Core

1.2.1 What type of applications can you build?

ASP.NET Core provides a generalized web framework that can be used for a variety of applications. It can most obviously be used for building rich, dynamic websites, whether they're e-commerce sites, content-based sites, or large n-tier applications—much the same as the previous version of ASP.NET.

When .NET Core was originally released, there were few third-party libraries available for building these types of complex applications. After several years of active development, that's no longer the case. Many developers have updated their libraries to work with ASP.NET Core, and many other libraries have been created to target ASP.NET Core specifically. For example, the open source content management system

(CMS) Orchard[1] has been redeveloped as Orchard Core[2] to run on ASP.NET Core. In contrast, the cloudscribe[3] CMS project (figure 1.5) was written specifically for ASP.NET Core from its inception.

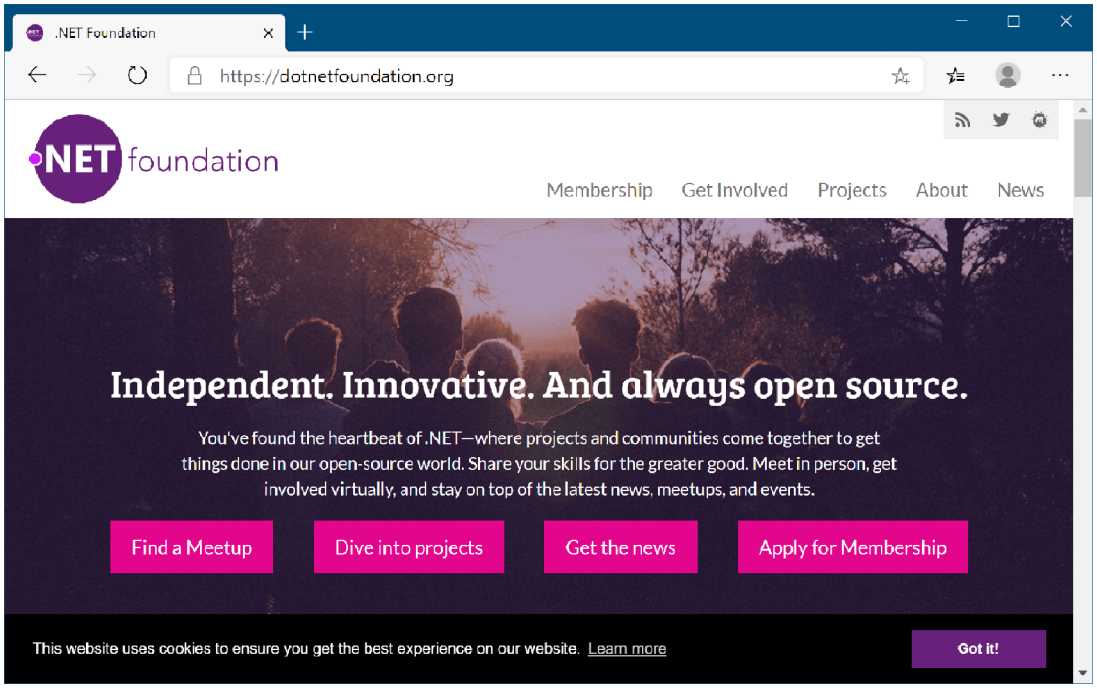

Figure 1.5 The .NET Foundation website (https://dotnetfoundation.org/) is built using the cloudscribe CMS and ASP.NET Core.

Traditional page-based server-side-rendered web applications are the bread and butter of ASP.NET development, both with the previous version of ASP.NET and with ASP.NET Core. Additionally, single-page applications (SPAs), which use a client-side framework that commonly talks to a REST server, are easy to create with ASP.NET Core. Whether you're using Angular, Vue, React, or some other client-side framework, it's easy to create an ASP.NET Core application to act as the server-side API.

> **DEFINITION** *REST* stands for *representational state transfer*. RESTful applications typically use lightweight and stateless HTTP calls to read, post (create/update), and delete data.

[1] The Orchard project source code is at https://github.com/OrchardCMS.
[2] Orchard Core (www.orchardcore.net). Source code at https://github.com/OrchardCMS/OrchardCore.
[3] The cloudscribe project (www.cloudscribe.com). Source code at https://github.com/cloudscribe.

ASP.NET Core isn't restricted to creating RESTful services. It's easy to create a web service or remote procedure call (RPC)-style service for your application, depending on your requirements, as shown in figure 1.6. In the simplest case, your application might expose only a single endpoint, narrowing its scope to become a microservice. ASP.NET Core is perfectly designed for building simple services, thanks to its cross-platform support and lightweight design.

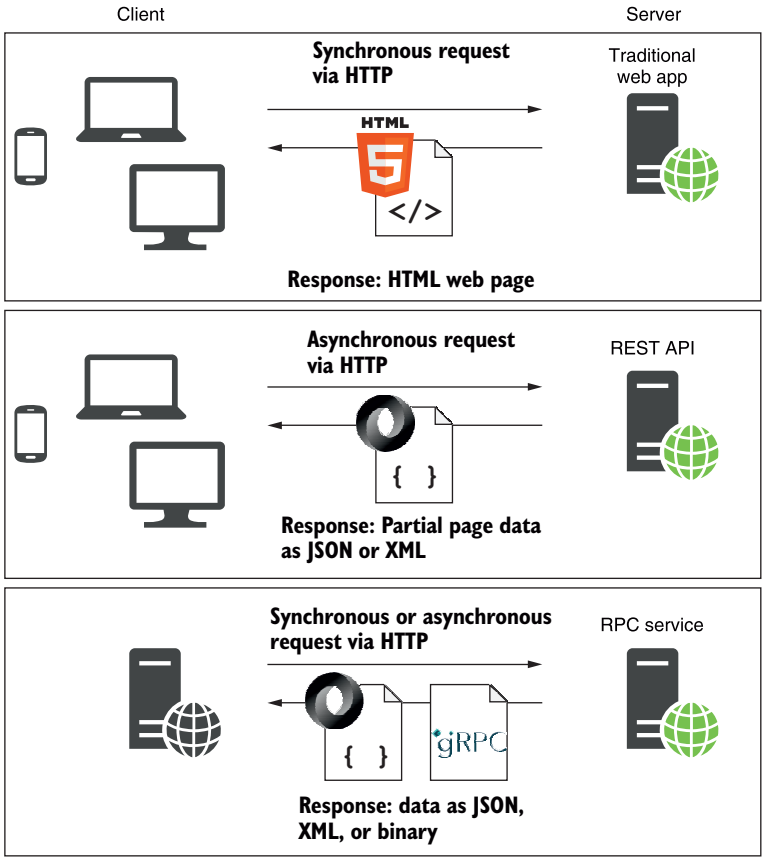

Figure 1.6 ASP.NET Core can act as the server-side application for a variety of clients: it can serve HTML pages for traditional web applications, it can act as a REST API for client-side SPA applications, or it can act as an ad hoc RPC service for client applications.

> **NOTE** In this book I focus on building traditional, page-based server-side-rendered web applications and RESTful web APIs. I also show how to create "headless" worker services in chapter 22.

You should consider multiple factors when choosing a platform, not all of which are technical. One such factor is the level of support you can expect to receive from its

creators. For some organizations, this can be one of the main obstacles to adopting open source software. Luckily, Microsoft has pledged to provide full support for Long Term Support (LTS) versions of .NET Core and ASP.NET Core for at least three years from the time of their release.[4] And as all development takes place in the open, you can sometimes get answers to your questions from the general community, as well as from Microsoft directly.

When deciding whether to use ASP.NET Core, you have two primary dimensions to consider: whether you're already a .NET developer, and whether you're creating a new application or looking to convert an existing one.

1.2.2 If you're new to .NET development

If you're new to .NET development and are considering ASP.NET Core, then welcome! Microsoft is pushing ASP.NET Core as an attractive option for web development beginners, but taking .NET cross-platform means it's competing with many other frameworks on their own turf. ASP.NET Core has many selling points when compared to other cross-platform web frameworks:

- It's a modern, high-performance, open source web framework.
- It uses familiar design patterns and paradigms.
- C# is a great language (or you can use VB.NET or F# if you prefer).
- You can build and run on any platform.

ASP.NET Core is a re-imagining of the ASP.NET framework, built with modern software design principles on top of the new .NET Core/.NET 5.0 platform. Although new in one sense, .NET Core has several years of widespread production use and has drawn significantly from the mature, stable, and reliable .NET Framework, which has been used for nearly two decades. You can rest easy knowing that by choosing ASP.NET Core and .NET 5.0, you'll be getting a dependable platform as well as a full-featured web framework.

Many of the web frameworks available today use similar well-established design patterns, and ASP.NET Core is no different. For example, Ruby on Rails is known for its use of the Model-View-Controller (MVC) pattern; Node.js is known for the way it processes requests using small discrete modules (called a *pipeline*); and dependency injection is found in a wide variety of frameworks. If these techniques are familiar to you, you should find it easy to transfer them across to ASP.NET Core; if they're new to you, then you can look forward to using industry best practices!

> **NOTE** You'll encounter a pipeline in chapter 3, MVC in chapter 4, and dependency injection in chapter 10.

The primary language of .NET development, and ASP.NET Core in particular, is C#. This language has a huge following, and for good reason! As an object-oriented C-based

[4] View the support policy at https://dotnet.microsoft.com/platform/support/policy/dotnet-core.

language, it provides a sense of familiarity to those used to C, Java, and many other languages. In addition, it has many powerful features, such as Language Integrated Query (LINQ), closures, and asynchronous programming constructs. The C# language is also designed in the open on GitHub, as is Microsoft's C# compiler, code-named Roslyn.[5]

> **NOTE** I use C# throughout this book and will highlight some of the newer features it provides, but I won't be teaching the language from scratch. If you want to learn C#, I recommend *C# in Depth*, fourth edition by Jon Skeet (Manning, 2019), and *Code like a Pro in C#* by Jort Rodenburg (Manning, 2021).

One of the major selling points of ASP.NET Core and .NET 5.0 is the ability to develop and run on any platform. Whether you're using a Mac, Windows, or Linux, you can run the same ASP.NET Core apps and develop across multiple environments. As a Linux user, a wide range of distributions are supported (RHEL, Ubuntu, Debian, CentOS, Fedora, and openSUSE, to name a few), so you can be confident your operating system of choice will be a viable option. ASP.NET Core even runs on the tiny Alpine distribution, for truly compact deployments to containers.

Built with containers in mind

Traditionally, web applications were deployed directly to a server, or more recently, to a virtual machine. Virtual machines allow operating systems to be installed in a layer of virtual hardware, abstracting away the underlying hardware. This has several advantages over direct installation, such as easy maintenance, deployment, and recovery. Unfortunately, they're also heavy, both in terms of file size and resource use.

This is where containers come in. Containers are far more lightweight and don't have the overhead of virtual machines. They're built in a series of layers and don't require you to boot a new operating system when starting a new one. That means they're quick to start and are great for quick provisioning. Containers, and Docker in particular, are quickly becoming the go-to platform for building large, scalable systems.

Containers have never been a particularly attractive option for ASP.NET applications, but with ASP.NET Core, .NET 5.0, and Docker for Windows, that's all changing. A lightweight ASP.NET Core application running on the cross-platform .NET 5.0 framework is perfect for thin container deployments. You can learn more about your deployment options in chapter 16.

As well as running on each platform, one of the selling points of .NET is the ability to write and compile only once. Your application is compiled to Intermediate Language (IL) code, which is a platform-independent format. If a target system has the .NET 5.0 runtime installed, you can run compiled IL from any platform. That means you can, for example, develop on a Mac or a Windows machine and deploy *the exact same files* to

[5] The C# language and .NET Compiler Platform GitHub source code repository can be found at https://github.com/dotnet/roslyn.

your production Linux machines. This compile-once, run-anywhere promise has finally been realized with ASP.NET Core and .NET Core/.NET 5.0.

1.2.3 If you're a .NET Framework developer creating a new application

If you're a .NET developer, the choice of whether to invest in ASP.NET Core for new applications has largely been a question of timing. Early versions of .NET Core were lacking in some features that made it hard to adopt. With the release of .NET Core 3.1 and .NET 5.0, that is no longer a problem; Microsoft now explicitly advises that all new .NET applications should use .NET 5.0. Microsoft has pledged to provide bug and security fixes for the older ASP.NET framework, but it won't provide any more feature updates. .NET Framework isn't being removed, so your old applications will continue to work, but you shouldn't use it for new development.

The main benefits of ASP.NET Core over the previous ASP.NET framework are

- Cross-platform development and deployment
- A focus on performance as a feature
- A simplified hosting model
- Regular releases with a shorter release cycle
- Open source
- Modular features

As a .NET developer, if you aren't using any Windows-specific constructs, such as the Registry, the ability to build and deploy cross-platform opens the door to a whole new avenue of applications: take advantage of cheaper Linux VM hosting in the cloud, use Docker containers for repeatable continuous integration, or write .NET code on your Mac without needing to run a Windows virtual machine. ASP.NET Core, in combination with .NET 5.0, makes all this possible.

.NET Core and .NET 5.0 are inherently cross-platform, but you can still use platform-specific features if you need to. For example, Windows-specific features like the Registry or Directory Services can be enabled with a compatibility pack that makes these APIs available in .NET 5.0.[6] They're only available when running .NET 5.0 on Windows, not on Linux or macOS, so you need to take care that such applications only run in a Windows environment or account for the potential missing APIs.

The hosting model for the previous ASP.NET framework was a relatively complex one, relying on Windows IIS to provide the web server hosting. In a cross-platform environment, this kind of symbiotic relationship isn't possible, so an alternative hosting model has been adopted—one that separates web applications from the underlying host. This opportunity has led to the development of Kestrel: a fast, cross-platform HTTP server on which ASP.NET Core can run.

Instead of the previous design, whereby IIS calls into specific points of your application, ASP.NET Core applications are console applications that self-host a web server

[6] The Windows Compatibility Pack is designed to help port code from .NET Framework to .NET Core/.NET 5.0. See https://docs.microsoft.com/dotnet/core/porting/windows-compat-pack.

and handle requests directly, as shown in figure 1.7. This hosting model is conceptually much simpler and allows you to test and debug your applications from the command

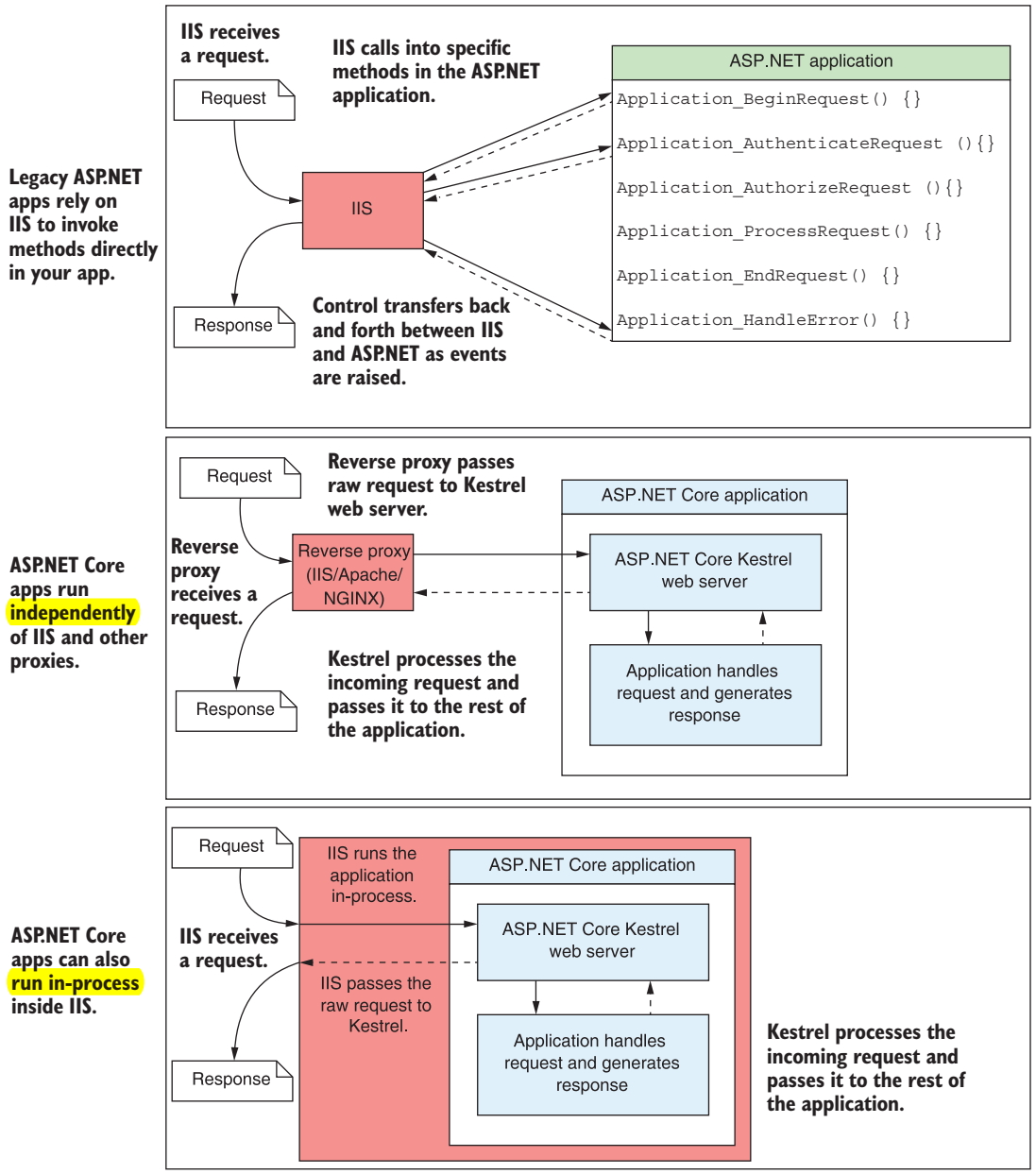

Figure 1.7 The difference between hosting models in ASP.NET (top) and ASP.NET Core (bottom). With the previous version of ASP.NET, IIS is tightly coupled with the application. The hosting model in ASP.NET Core is simpler; IIS hands off the request to a self-hosted web server in the ASP.NET Core application and receives the response, but has no deeper knowledge of the application.

line, though it doesn't necessarily remove the need to run IIS (or equivalent) in production, as you'll see in section 1.3.

> **NOTE** You can also optionally run ASP.NET Core *inside* of IIS, as shown in figure 1.7, which can have performance benefits over the reverse-proxy version. This is primarily a deployment detail and doesn't change the way you build ASP.NET Core applications.

Changing the hosting model to use a built-in HTTP web server has created another opportunity. Performance has been somewhat of a sore point for ASP.NET applications in the past. It's certainly possible to build high-performing applications—Stack Overflow (https://stackoverflow.com) is a testament to that—but the web framework itself isn't designed with performance as a priority, so it can end up being somewhat of an obstacle.

To be competitive cross-platform, the ASP.NET team has focused on making the Kestrel HTTP server as fast as possible. TechEmpower (www.techempower.com/benchmarks) has been running benchmarks on a whole range of web frameworks from various languages for several years now. In Round 19 of the plain text benchmarks, TechEmpower announced that ASP.NET Core with Kestrel was the fastest of over 400 frameworks tested![7]

Web servers: naming things is hard

One of the difficult aspects of programming for the web is the confusing array of often conflicting terminology. For example, if you've used IIS in the past, you may have described it as a web server, or possibly a web host. Conversely, if you've ever built an application using Node.js, you may have also referred to that application as a web server. Alternatively, you may have called the physical machine on which your application runs a web server!

Similarly, you may have built an application for the internet and called it a website or a web application, probably somewhat arbitrarily based on the level of dynamism it displayed.

In this book, when I say "web server" in the context of ASP.NET Core, I am referring to the HTTP server that runs as part of your ASP.NET Core application. By default, this is the Kestrel web server, but that's not a requirement. It would be possible to write a replacement web server and substitute it for Kestrel if you desired.

The web server is responsible for receiving HTTP requests and generating responses. In the previous version of ASP.NET, IIS took this role, but in ASP.NET Core, Kestrel is the web server.

I will only use the term "web application" to describe ASP.NET Core applications in this book, regardless of whether they contain only static content or are completely dynamic. Either way, they're applications that are accessed via the web, so that name seems the most appropriate.

[7] As always in web development, technology is in a constant state of flux, so these benchmarks will evolve over time. Although ASP.NET Core may not maintain its top-ten slot, you can be sure that performance is one of the key focal points of the ASP.NET Core team.

Many of the performance improvements made to Kestrel did not come from the ASP.NET team members themselves, but from contributors to the open source project on GitHub.[8] Developing in the open means you typically see fixes and features make their way to production faster than you would for the previous version of ASP.NET, which was dependent on .NET Framework and Windows and, as such, had long release cycles.

In contrast, .NET 5.0, and hence ASP.NET Core, is designed to be released in small increments. Major versions will be released on a predictable cadence, with a new version every year, and a new Long Term Support (LTS) version released every two years.[9] In addition, bug fixes and minor updates can be released as and when they're needed. Additional functionality is provided as NuGet packages, independent of the underlying .NET 5.0 platform.

> **NOTE** NuGet is a package manager for .NET that enables importing libraries into your projects. It's equivalent to Ruby Gems, npm for JavaScript, or Maven for Java.

To enable this approach to releases, ASP.NET Core is highly modular, with as little coupling to other features as possible. This modularity lends itself to a pay-for-play approach to dependencies, where you start with a bare-bones application and only add the additional libraries you require, as opposed to the kitchen-sink approach of previous ASP.NET applications. Even MVC is an optional package! But don't worry, this approach doesn't mean that ASP.NET Core is lacking in features; it means you need to opt in to them. Some of the key infrastructure improvements include

- Middleware "pipeline" for defining your application's behavior
- Built-in support for dependency injection
- Combined UI (MVC) and API (Web API) infrastructure
- Highly extensible configuration system
- Scalable for cloud platforms by default using asynchronous programming

Each of these features was possible in the previous version of ASP.NET but required a fair amount of additional work to set up. With ASP.NET Core, they're all there, ready, and waiting to be connected!

Microsoft fully supports ASP.NET Core, so if you have a new system you want to build, there's no significant reason not to use it. The largest obstacle you're likely to come across is wanting to use programming models that are no longer supported in ASP.NET Core, such as Web Forms or WCF server, as I'll discuss in the next section.

Hopefully, this section has whetted your appetite with some of the many reasons to use ASP.NET Core for building new applications. But if you're an existing ASP.NET

---

[8] The Kestrel HTTP server GitHub project can be found in the ASP.NET Core repository at https://github.com/dotnet/aspnetcore.

[9] The release schedule for .NET 5.0 and beyond: https://devblogs.microsoft.com/dotnet/introducing-net-5/.

developer considering whether to convert an existing ASP.NET application to ASP.NET Core, that's another question entirely.

1.2.4 Converting an existing ASP.NET application to ASP.NET Core

In contrast with new applications, an existing application is presumably already providing value, so there should always be a tangible benefit to performing what may amount to a significant rewrite in converting from ASP.NET to ASP.NET Core. The advantages of adopting ASP.NET Core are much the same as for new applications: cross-platform deployment, modular features, and a focus on performance. Whether the benefits are sufficient will depend largely on the particulars of your application, but there are some characteristics that are clear indicators *against* conversion:

- Your application uses ASP.NET Web Forms
- Your application is built using WCF
- Your application is large, with many "advanced" MVC features

If you have an ASP.NET Web Forms application, attempting to convert it to ASP.NET Core isn't advisable. Web Forms is inextricably tied to System.Web.dll, and as such will likely never be available in ASP.NET Core. Converting an application to ASP.NET Core would effectively involve rewriting the application from scratch, not only shifting frameworks but also shifting design paradigms. A better approach would be to slowly introduce Web API concepts and try to reduce the reliance on legacy Web Forms constructs such as ViewData. You can find many resources online to help you with this approach, in particular the www.asp.net/web-api website.[10]

Windows Communication Foundation (WCF) is only partially supported in ASP.NET Core.[11] It's possible to consume some WCF services, but support is spotty at best. There's no supported way to host a WCF service from an ASP.NET Core application, so if you absolutely must support WCF, then ASP.NET Core may be best avoided for now.

> **TIP** If you like WCFs RPC-style of programming, but you don't have a hard requirement on WCF itself, consider using gRPC instead. gRPC is a modern RPC framework with many concepts similar to WCF, and it's supported by ASP.NET Core out of the box.[12]

If your existing application is complex and makes extensive use of the previous MVC or Web API extensibility points or message handlers, then porting your application to ASP.NET Core may be more difficult. ASP.NET Core is built with many similar features to the previous version of ASP.NET MVC, but the underlying architecture is

---

[10] An alternative approach would be to consider converting your application to Blazor using the community-driven effort to create Blazor versions of common WebForms components: https://github.com/FritzAndFriends/BlazorWebFormsComponents.

[11] You can find the client libraries for using WCF with .NET Core at https://github.com/dotnet/wcf.

[12] You can find an eBook from Microsoft on gRPC for WCF developers at https://docs.microsoft.com/en-us/dotnet/architecture/grpc-for-wcf-developers/.

different. Several of the previous features don't have direct replacements and so will require rethinking.

The larger the application, the greater the difficulty you're likely to have converting your application to ASP.NET Core. Microsoft itself suggests that porting an application from ASP.NET MVC to ASP.NET Core is at least as big a rewrite as porting from ASP.NET Web Forms to ASP.NET MVC. If that doesn't scare you, then nothing will!

If an application is rarely used, isn't part of your core business, or won't need significant development in the near term, I strongly suggest you *don't* try to convert it to ASP.NET Core. Microsoft will support .NET Framework for the foreseeable future (Windows itself depends on it!), and the payoff in converting these "fringe" applications is unlikely to be worth the effort.

So, when *should* you port an application to ASP.NET Core? As I've already mentioned, the best opportunity for getting started is on small, green-field, new projects instead of existing applications. That said, if the existing application in question is small or will need significant future development, then porting may be a good option. It is always best to work in small iterations where possible, rather than attempting to convert the entire application at once. But if your application consists primarily of MVC or Web API controllers and associated Razor views, moving to ASP.NET Core may well be a good choice.

1.3 How does ASP.NET Core work?

By now, you should have a good idea of what ASP.NET Core is and the sort of applications you should use it for. In this section, you'll see how an application built with ASP.NET Core works, from the user requesting a URL to a page being displayed in the browser. To get there, first you'll see how an HTTP request works for any web server, and then you'll see how ASP.NET Core extends the process to create dynamic web pages.

1.3.1 How does an HTTP web request work?

As you know, ASP.NET Core is a framework for building web applications that serve data from a server. One of the most common scenarios for web developers is building a web app that you can view in a web browser. The high-level process you can expect from any web server is shown in figure 1.8.

The process begins when a user navigates to a website or types a URL in their browser. The URL or web address consists of a *hostname* and a *path* to some resource on the web app. Navigating to the address in the browser sends a request from the user's computer to the server on which the web app is hosted, using the HTTP protocol.

DEFINITION The *hostname* of a website uniquely identifies its location on the internet by mapping via the Domain Name Service (DNS) to an IP address. Examples include microsoft.com, www.google.co.uk, and facebook.com.

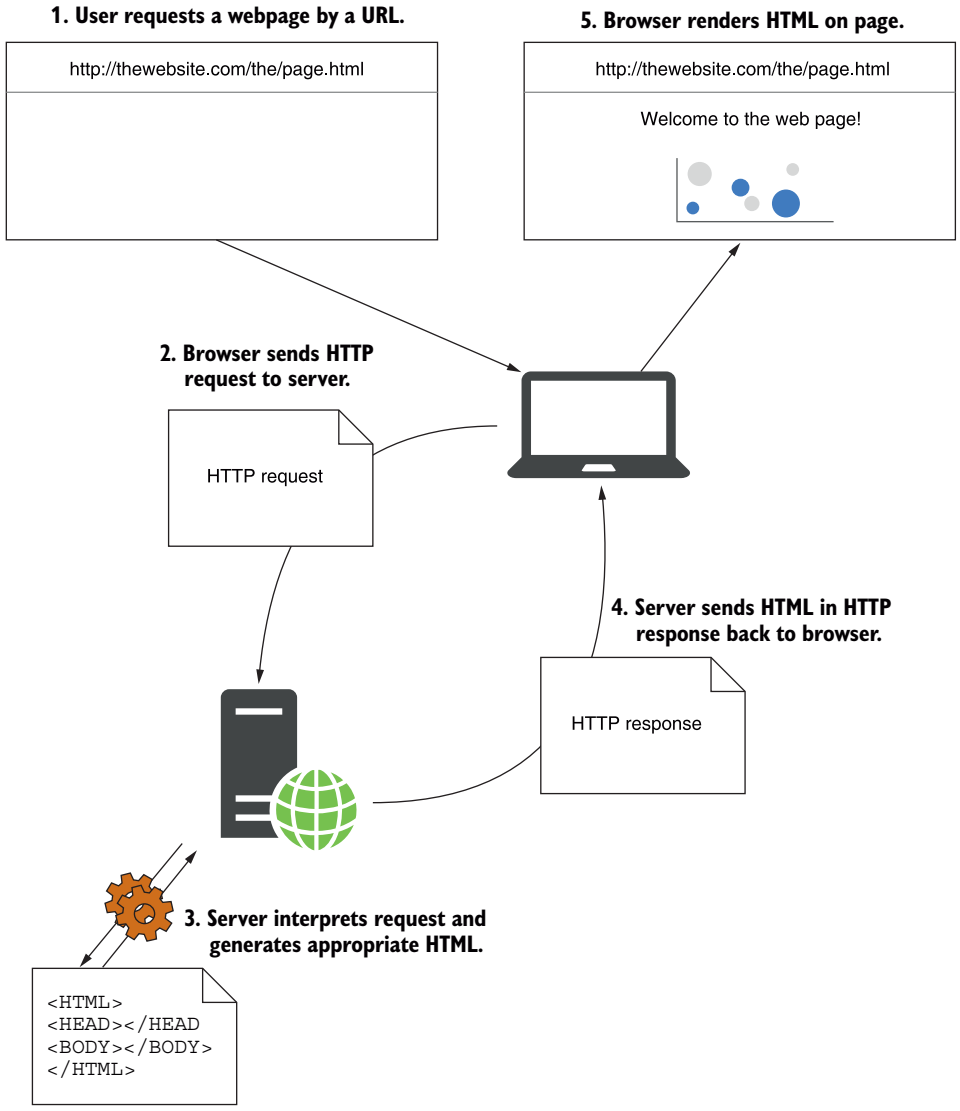

Figure 1.8 Requesting a web page. The user starts by requesting a web page, which causes an HTTP request to be sent to the server. The server interprets the request, generates the necessary HTML, and sends it back in an HTTP response. The browser can then display the web page.

The request passes through the internet, potentially to the other side of the world, until it finally makes its way to the server associated with the given hostname, on which the web app is running. The request is potentially received and rebroadcast at multiple routers along the way, but it's only when it reaches the server associated with the hostname that the request is processed.

A brief primer on HTTP

Hypertext Transfer Protocol (HTTP) is the application-level protocol that powers the web. It is a stateless request-response protocol, whereby a client machine sends a *request* to a server, which sends a *response* in turn.

Every HTTP request consists of a *verb* indicating the "type" of the request and a *path* indicating the resource to interact with. They typically also include *headers*, which are key-value pairs, and in some cases a *body*, such as the contents of a form, when sending data to the server.

An HTTP response contains a *status code*, indicating whether the request was successful, and optionally *headers* and a *body*.

For a more detailed look at the HTTP protocol itself, as well as more examples, see section 1.3 ("A quick introduction to HTTP") in *Go Web Programming* by Sau Sheong Chang (Manning, 2016), https://livebook.manning.com/book/go-web-programming/chapter-1/point-9018-55-145-1.

Once the server receives the request, it will check that the request makes sense, and if it does, it will generate an HTTP response. Depending on the request, this response could be a web page, an image, a JavaScript file, or a simple acknowledgment. For this example, I'll assume the user has reached the home page of a web app, so the server responds with some HTML. The HTML is added to the HTTP response, which is then sent back across the internet to the browser that made the request.

As soon as the user's browser begins receiving the HTTP response, it can start displaying content on the screen, but the HTML page may also reference other pages and links on the server. To display the complete web page, instead of a static, colorless, raw HTML file, the browser must repeat the request process, fetching every referenced file. HTML, images, CSS for styling, and JavaScript files for extra behavior are all fetched using the exact same HTTP request process.

Pretty much all interactions that take place on the internet are a facade over this same basic process. A basic web page may only require a few simple requests to fully render, whereas a modern, large web page may take hundreds. At the time of writing, the Amazon.com homepage (www.amazon.com), for example, makes 606 requests, including ones for 3 CSS files, 12 JavaScript files, and 402 image files!

Now that you have a feel for the process, let's see how ASP.NET Core dynamically generates the response on the server.

1.3.2 How does ASP.NET Core process a request?

When you build a web application with ASP.NET Core, browsers will still be using the same HTTP protocol as before to communicate with your application. ASP.NET Core itself encompasses everything that takes place on the server to handle a request, including verifying that the request is valid, handling login details, and generating HTML.

Just as with the generic web page example, the request process starts when a user's browser sends an HTTP request to the server, as shown in figure 1.9.

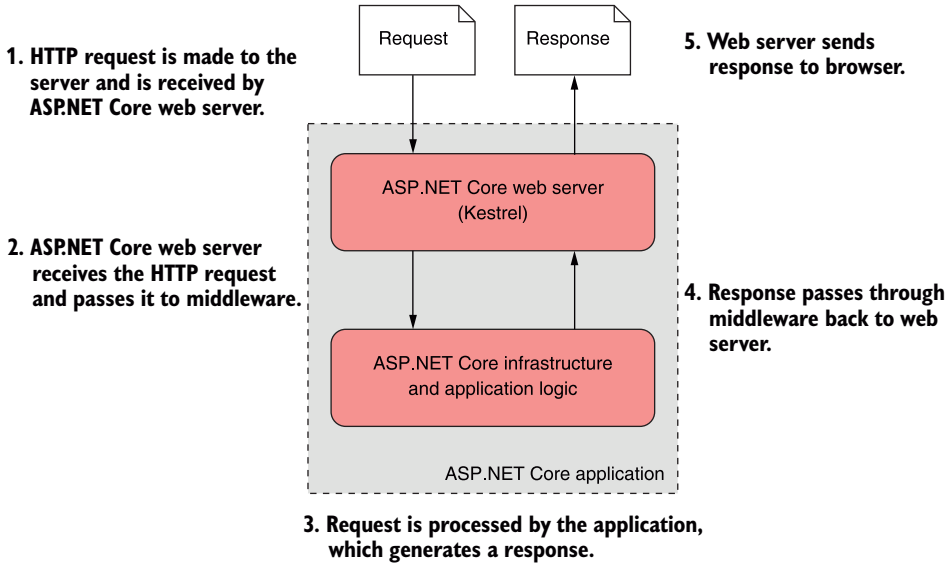

Figure 1.9 How an ASP.NET Core application processes a request. A request is received by the ASP.NET Core application, which runs a self-hosted web server. The web server processes the request and passes it to the body of the application, which generates a response and returns it to the web server. The web server sends this response to the browser.

The request is received from the network by your ASP.NET Core application. Every ASP.NET Core application has a built-in web server, Kestrel by default, which is responsible for receiving raw requests and constructing an internal representation of the data, an `HttpContext` object, which can be used by the rest of the application.

From this representation, your application should have all the details it needs to create an appropriate response to the request. It can use the details stored in `HttpContext` to generate an appropriate response, which may be to generate some HTML, to return an "access denied" message, or to send an email, all depending on your application's requirements.

Once the application has finished processing the request, it will return the response to the web server. The ASP.NET Core web server will convert the representation into a raw HTTP response and send it to the network, which will forward it to the user's browser.

To the user, this process appears to be the same as for the generic HTTP request shown in figure 1.8—the user sent an HTTP request and received an HTTP response. All the differences are server-side, within your application.

ASP.NET Core and reverse proxies

You can expose ASP.NET Core applications directly to the internet, so that Kestrel receives requests directly from the network. However, it's more common to use a reverse proxy between the raw network and your application. In Windows, the reverse-proxy server will typically be IIS, and on Linux or macOS it might be NGINX, HAProxy, or Apache.

A *reverse proxy* is software responsible for receiving requests and forwarding them to the appropriate web server. The reverse proxy is exposed directly to the internet, whereas the underlying web server is exposed only to the proxy. This setup has several benefits, primarily security and performance for the web servers.

You may be thinking that having a reverse proxy *and* a web server is somewhat redundant. Why not have one or the other? Well, one of the benefits is the decoupling of your application from the underlying operating system. The same ASP.NET Core web server, Kestrel, can be cross-platform and used behind a variety of proxies without putting any constraints on a particular implementation. Alternatively, if you wrote a new ASP.NET Core web server, you could use that in place of Kestrel without needing to change anything else about your application.

Another benefit of a reverse proxy is that it can be hardened against potential threats from the public internet. They're often responsible for additional aspects, such as restarting a process that has crashed. Kestrel can remain a simple HTTP server not having to worry about these extra features when it's used behind a reverse proxy. Think of it as a simple separation of concerns: Kestrel is concerned with generating HTTP responses; the reverse proxy is concerned with handling the connection to the internet.

You've seen how requests and responses find their way to and from an ASP.NET Core application, but I haven't yet touched on how the response is generated. In part 1 of this book, we'll look at the components that make up a typical ASP.NET Core application and how they all fit together. A lot goes into generating a response in ASP.NET Core, typically all within a fraction of a second, but over the course of the book we'll step through an application slowly, covering each of the components in detail.

1.4 What you will learn in this book

This book will take you on an in-depth tour of the ASP.NET Core framework. To benefit from the book, you should be familiar with C# or a similar objected-oriented language. Basic familiarity with web concepts like HTML and JavaScript will also be beneficial. You will learn

- How to create page-based applications with Razor Pages
- Key ASP.NET Core concepts like model-binding, validation, and routing
- How to generate HTML for web pages using Razor syntax and Tag Helpers
- To use features like dependency injection, configuration, and logging as your applications grow more complex
- How to protect your application using security best practices

Throughout the book we'll use a variety of examples to learn and explore concepts. The examples are generally small and self-contained so we can focus on a single feature at a time.

I'll be using Visual Studio for most of the examples in this book, but you'll be able to follow along using your favorite editor or IDE. Appendix A includes details on setting up your editor or IDE and installing the .NET 5.0 SDK. Even though the examples in this book show Windows tools, everything you see can be achieved equally well on Linux or Mac platforms.

> **TIP** You can install .NET 5.0 from https://dotnet.microsoft.com/download. Appendix A contains further details on how to configure your development environment for working with ASP.NET Core and .NET 5.0.

In the next chapter, you'll create your first application from a template and run it. We'll walk through each of the main components that make up your application and see how they all work together to render a web page.

Summary

- ASP.NET Core is a new web framework built with modern software architecture practices and modularization as its focus.
- It's best used for new, "green-field" projects.
- Legacy technologies such as WCF Server and Web Forms can't be used with ASP.NET Core.
- ASP.NET Core runs on the cross-platform .NET 5.0 platform. You can access Windows-specific features such as the Windows Registry by using the Windows Compatibility Pack.
- .NET 5.0 is the next version of .NET Core after .NET Core 3.1.
- Fetching a web page involves sending an HTTP request and receiving an HTTP response.
- ASP.NET Core allows you to dynamically build responses to a given request.
- An ASP.NET Core application contains a web server, which serves as the entry point for a request.
- ASP.NET Core apps are typically protected from the internet by a reverse-proxy server, which forwards requests to the application.